<https://leetcode.com/problems/sort-colors/>

Given an array nums with n objects colored red, white, or blue, sort them **[in-place](https://en.wikipedia.org/wiki/In-place_algorithm)**so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

**Example 1:**

Input: nums = [2,0,2,1,1,0]

Output: [0,0,1,1,2,2]

**Example 2:**

Input: nums = [2,0,1]

Output: [0,1,2]

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* nums[i] is either 0, 1, or 2.

**Follow up:** Could you come up with a one-pass algorithm using only constant extra space?

**Attempt 1: 2023-02-28**

**Solution 1: Three Pointers (30 min)**

class Solution {

public void sortColors(int[] nums) {

// 0, 1, and 2 to represent the color red, white, and blue

int red = 0;

int blue = nums.length - 1;

int white = 0;

while(white <= blue) {

if(nums[white] == 0) {

swap(nums, white, red);

red++;

white++;

} else if(nums[white] == 2) {

swap(nums, white, blue);

blue--;

} else {

white++;

}

}

}

private void swap(int[] nums, int i, int j) {

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

}

Time Complexity:O(n)

Space Complexity:O(1)

**Why don't you increment white pointer when you swap with blue pointer? (else case)**

**Refer to**

<https://leetcode.com/problems/sort-colors/solutions/26481/python-o-n-1-pass-in-place-solution-with-explanation/comments/1129703>

Because you might have swapped a red color for blue color, so in the next iteration the red color will be swapped for white color. Giving you the correct order.

What a lot of people are indirectly referring to but don't explicitly mention is that there's an *invariant* at play here: we assume all the elements to the left of white to be color-classified and all the elements to the right (and including) white to be color-unclassified. Swapping white with red (a pointer that, for the most part, is to the left) leaves the element at white as color-classified, hence we can increment it. On the other hand, swapping white with blue (a pointer that, for the most part, is to the right) leaves the element at white as color-unclassified (as the element that white is now pointing to used to be pointed to by blue), hence we cannot increment it.

**Refer to**

<https://leetcode.com/problems/sort-colors/solutions/26481/python-o-n-1-pass-in-place-solution-with-explanation/>

0, 1, and 2 to represent the color red, white, and blue

This is a [dutch partitioning problem](https://en.wikipedia.org/wiki/Dutch_national_flag_problem). We are classifying the array into four groups: red, white, unclassified, and blue. Initially we group all elements into unclassified. We iterate from the beginning as long as the white pointer is less than the blue pointer.

If the white pointer is red (nums[white] == 0), we swap with the red pointer and move both white and red pointer forward. If the pointer is white (nums[white] == 1), the element is already in correct place, so we don't have to swap, just move the white pointer forward. If the white pointer is blue, we swap with the latest unclassified element.

def sortColors(self, nums):

red, white, blue = 0, 0, len(nums)-1

while white <= blue:

if nums[white] == 0:

nums[red], nums[white] = nums[white], nums[red]

white += 1

red += 1

elif nums[white] == 1:

white += 1

else:

nums[white], nums[blue] = nums[blue], nums[white]

blue -= 1

**Refer to**

<https://leetcode.com/problems/sort-colors/solutions/26474/sharing-c-solution-with-good-explanation/>

The solution requires the use of tracking 3 positions, the Low, Mid and High.

We assume that the mid is the "Unknown" area that we must evaluate.

If we encounter a 0, we know that it will be on the low end of the array, and if we encounter a 2, we know it will be on the high end of the array.

To achieve this in one pass without preprocessing (counting), we simply traverse the unknown will generating the low and high ends.

Take this example:

Assume our input is: 1 0 2 2 1 0 (short for simplicity).

Running the algorithm by hand would look something like:

1 0 2 2 1 0

^ ^

L H

M

Mid != 0 || 2

Mid++

1 0 2 2 1 0

^ ^ ^

L M H

Mid == 0

Swap Low and Mid

Mid++

Low++

0 1 2 2 1 0

^ ^ ^

L M H

Mid == 2

Swap High and Mid

High--

0 1 0 2 1 2

^ ^ ^

L M H

Mid == 0

Swap Low and Mid

Mid++

Low++

0 0 1 2 1 2

^ ^ ^

L M H

Mid == 2

Swap High and Mid

High--

0 0 1 1 2 2

^ ^

L M

H

Mid <= High is our exit case

Implemented in C++, it looks like:

class Solution {

public:

void sortColors(vector<int>& nums)

{

int tmp = 0, low = 0, mid = 0, high = nums.size() - 1;

while(mid <= high)

{

if(nums[mid] == 0)

{

tmp = nums[low];

nums[low] = nums[mid];

nums[mid] = tmp;

low++;

mid++;

}

else if(nums[mid] == 1)

{

mid++;

}

else if(nums[mid] == 2)

{

tmp = nums[high];

nums[high] = nums[mid];

nums[mid] = tmp;

high--;

}

}

}

};